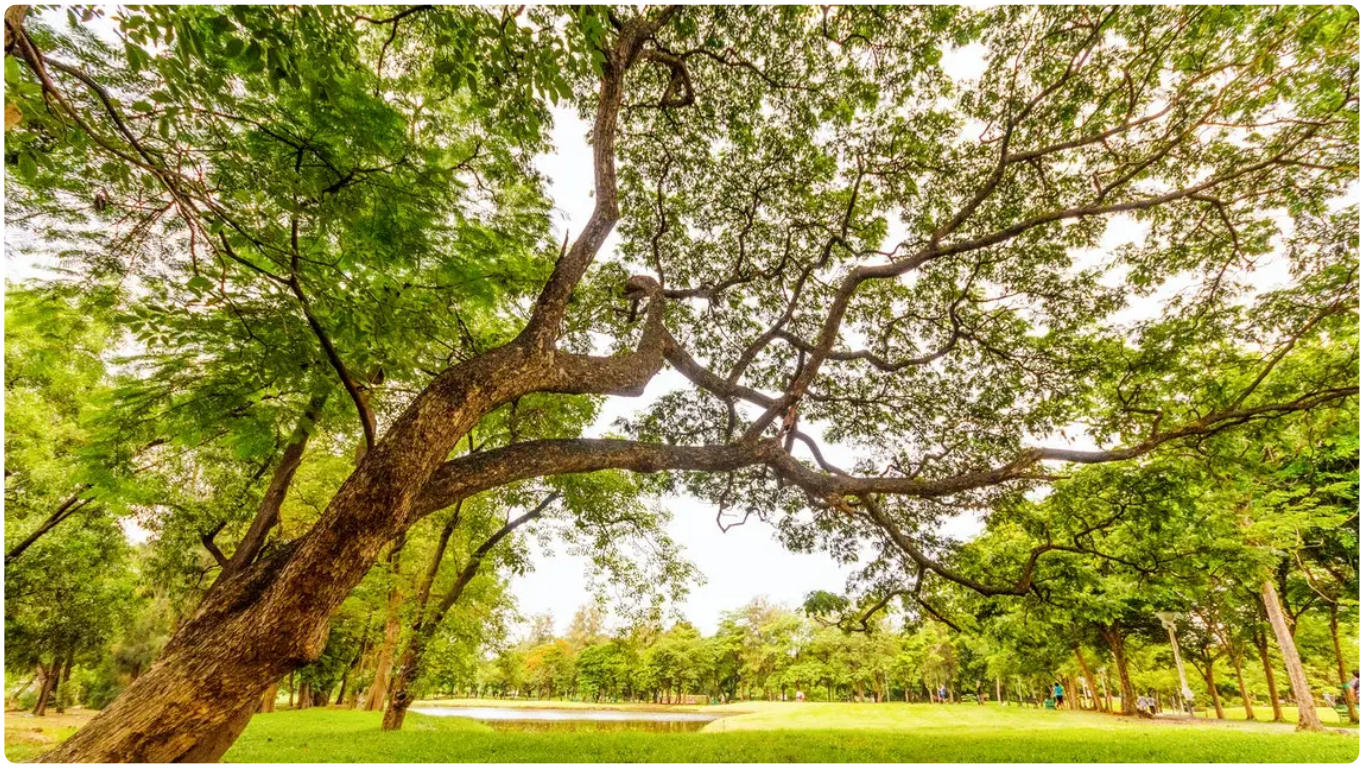
课前热身｜学习这门课前，你需要了解哪些知识？

2021-12-06 于航

《深入C语言和程序运行原理》 课程介绍 >



讲述：于航
时长 11:26 大小 10.48M

你好，我是于航。

在我们正式进入到 C 语言课程的学习之前，为了帮助你更好地理解课程内容，我为你准备了一节基础知识讲解课。这是一节选学课，你可以根据自己的实际情况选择性学习。

在这一讲中，我会用通俗易懂的方式，为你介绍这门课中最常用的一些基础知识，分别是常见数据量单位、汇编语言，以及 CPU 指令集中涉及的不同类型的寄存器。如果你对这些内容还不太熟悉，那么通过这一讲，你可以对它们有一个大致的印象。我们后面的课程还会提到汇编指令或寄存器，我会视情况进行更加具体的讲解，帮你加深理解。

数据量单位：位、字节和字

位（bit）是计算机中最小的存储单位，每一个位可以存储一个二进制码值的 0 或 1。而字节（byte）则通常是由八个位组成的一个存储单元。在计算机中，字节是最小的可寻址单位，这意味着 CPU 在使用数据时，可以以字节为单位，为每一字节内存分配一个相应的独立地址。

位和字节是在我们的日常工作中最为常见的两个数据量概念，你应该很熟悉。不过，字（word）的概念就没有这么清晰了。字的大小并不固定，一个字的大小可能是 2 的幂次个位，比如 16 位、32 位，也有可能是 12 位、27 位等一些并不常见的大小。而这主要是因为字的概念与具体的处理器或硬件体系架构直接相关，它跟位、字节这种较为通用和统一的数据量概念并不相同。

字是处理器设计时使用的自然数据单位，通常，这个大小会反映在计算机结构和相关操作的多个方面中。比如，处理器中大多数寄存器的容量是与字同样大小的，处理器单个指令可以操作的最大内存块一般为一个字大小，而用于指定内存中某个具体位置的地址，一般也是以处理器的自然字为宽度的。

需要说明下，在这门课后面的内容中，**所有给出的示例代码和相关分析，都是在平台类型为 x86-64 的实验机上进行的，而该平台的字长为 64 位。**

汇编语言

在计算机编程中，汇编语言（Assembly Language）是一种低级编程语言，语言使用的指令与具体平台紧密相关。这意味着，针对不同 CPU 体系架构设计的汇编语言无法共用，也不具备可移植性。

汇编代码可以经由汇编程序（如 as）进行转换，从而得到二进制的可执行代码。不同于高级编程语言，汇编语言在机器指令之上基本不具有任何抽象。因此，通过观察一个程序的汇编代码，我们可以详细了解到程序运行时的每一个具体步骤。所以，**在这门课中，我们会通过汇编代码来观察 C 语法的实现细节，并同时探索程序运行时与操作系统交互的一些关键步骤。**

汇编语言使用助记符（Mnemonic）来表示每个低级的机器指令。助记符是一类带有自然语义的符号，比如 `mov` 指令，它是英文单词 “move” 的简写，这个指令用来将一个操作数从计算机中的某个位置移动到另一个位置。而 `add` 指令的语义就更加直观了，它用来将计算机中某个位置上的数据量累加到另一个数据量上。

不同的汇编指令可以使用不同的参数形式。比如，就 mov 指令来说，对于可以使用该指令来移动的数据，它们通常会位于计算机中三个不同的位置上：

MOV r/m, r

MOV r, r/m

MOV r/m, imm

在这些指令的参数中，r 表示 register，即寄存器；m 表示 memory，即内存中的某个具体位置；imm 表示 immediate，即直接书写在指令中的立即数。

为了跟这门课后续的内容保持一致，这里我们直接使用 x86-64 平台，并基于 Intel 指令集的方式来书写和解读汇编代码。因此，指令 `mov ebx, 1` 的正确含义是：将立即数 1 存放到寄存器 *ebx* 中（右侧参数为数据来源 src，左侧参数为移动的目的地 dest）。需要注意的是，在 x86 指令集中，受限于 CPU 实现的复杂度，不存在可以将两个内存地址同时作为 src 和 dest 参数的指令。

汇编指令由助记符组成，而汇编器则负责把这些助记符组成的有效语法格式转换成对应的二进制机器指令。比如就上面提到的汇编指令 `mov ebx, 1` 而言，它所对应的机器指令代码为二进制值 `bb 01 00 00 00`。

与汇编代码不同的是，二进制机器指令代码的组成结构要复杂许多。上面的汇编指令对应的机器指令是由 OpCode 和 Immediate Data 两部分组成的。OpCode 在这里占用一个字节，这个字节是由指令对应的 0xb8 ，外加特定目的寄存器对应的寄存器域值 0x3 组成的。而紧跟着 OpCode 的，便是立即数 1 对应的部分。由于该指令用于传送 32 位数，因此立即数这里单独占用 4 个字节。需要注意的是，对于这段机器指令代码，最左侧的字节 0xbb 处在内存的低位（即小端序）。

在较为复杂的机器指令中，还可能包含有与 ModR/M、SIB、Displacement 以及 REX 等有关的信息。而这些信息有些并不会直接体现在上层的汇编代码中，它们大多仅与当前平台 CPU 的体系架构，或操作系统所处的模式有关。

指令集中的寄存器

在编写汇编语言代码时，我们经常会跟寄存器打交道。那么，什么是寄存器呢？

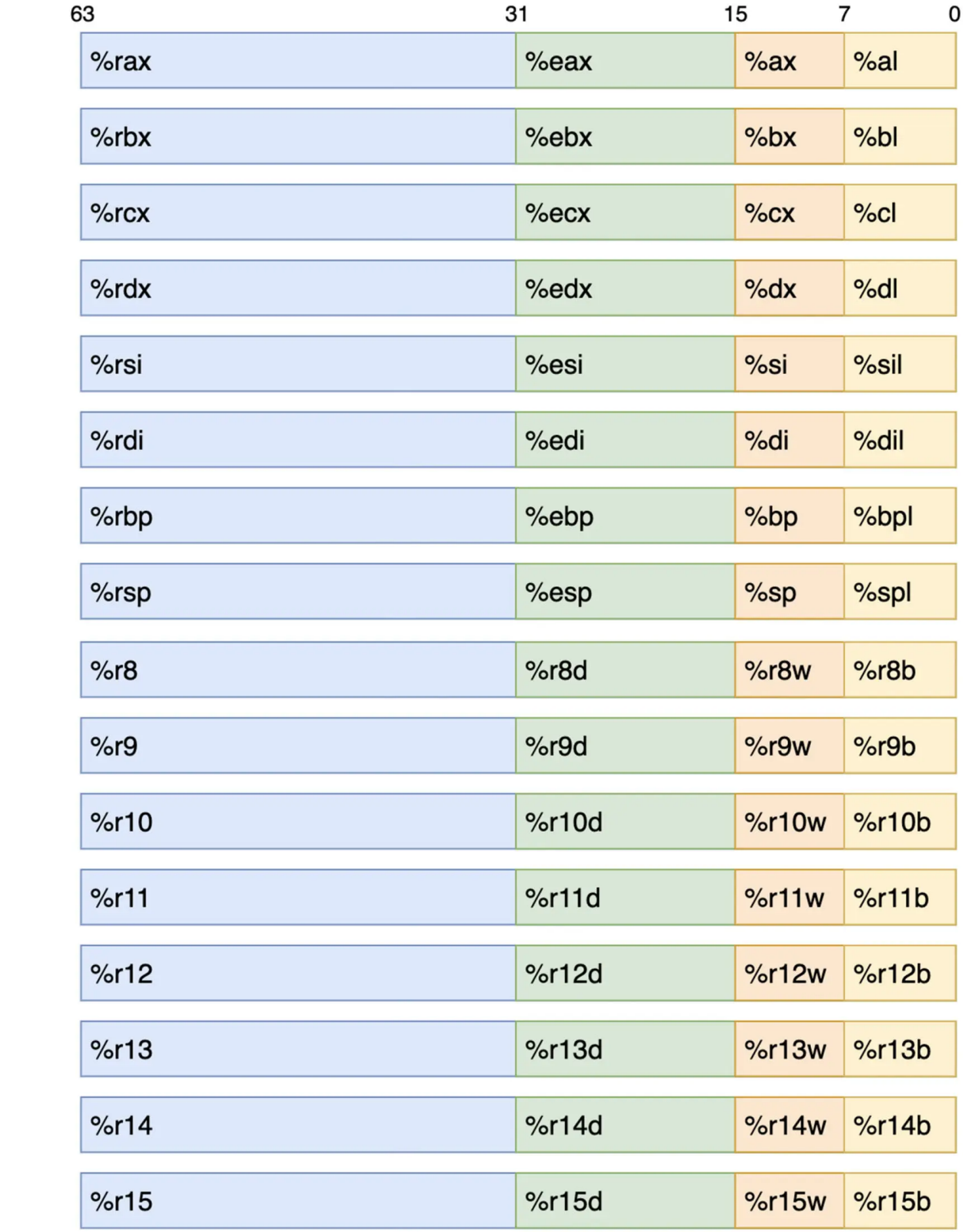
寄存器有时也被称为“寄存器文件（Register File）”，你可以把它简单理解为由 CPU 提供的一组位于芯片上的高速存储器硬件，可用于存储数据。通常来说，寄存器可以使用 SRAM 来实现。SRAM 是一种高速随机访问存储器，它将每个位的数据存放在一个对应的“双稳态”存储器中，从而保持较强的抗干扰能力和较快的数据访问速度。在整个计算机体系架构中，**寄存器拥有最快的数据访问速度和最低的延迟**。

通常来说，我们在汇编代码中使用的寄存器（比如之前提到的 *ebx*）可能并不与 CPU 上的物理寄存器完全一一对应，CPU 会使用额外的方式来保证它们之间的动态对应关系。这些参与到程序运行过程的寄存器，一般可以分为：通用目的寄存器、状态寄存器、系统寄存器，以及用于支持浮点数计算和 SIMD 的 AVX、SSE 寄存器等。

在这些寄存器中，通用目的寄存器一般用于存放程序运行过程中产生的临时数据，这些寄存器在大多数情况下都可以被当作普通寄存器使用。而在某些特殊情况下，它们可能会被用于存放指令计算结果、系统调用号，以及与栈帧相关的内存地址等信息。状态寄存器一般用于存放与指令执行结果相关的状态信息，比如指令执行是否引起进位、计算结果是否为 0 等。系统寄存器一般由操作系统使用，这些寄存器描述了与虚拟内存、中断、CPU 模式等有关的信息。

在 x86-64 架构下，CPU 指令集架构（ISA）中一共定义了 16 个通用目的寄存器。这些寄存器最大可以存放 4 个指令字，即 64 位长的数据。需要注意的是，**这里我们提到的“指令字”与之前介绍的用于描述 CPU 硬件特征的“硬件字”有所不同**（指令字与硬件字这两个叫法只是我用来区分这两种字概念的）。由于历史原因，在现代 x86 系列 CPU 的指令集文档中，你可能会看到对 WORD 一词的使用。虽然这个单词可以被翻译为“字”，但在这样的环境下，它实则代表着固定 16 位的长度。关于它的具体使用方式，我会在 03 讲 中详细介绍。

在汇编代码中，我们可以使用每个寄存器不同部分对应的别名，来针对性地访问它们的低 8 位、低 16 位、低 32 位，以及完整的 64 位数据。关于这些寄存器的具体名称，你可以参考下面这张图：



这张图怎么看呢？这里以我们之前遇到的 *ebx* 寄存器为例：观察上图可以得知，通过 *ebx*，我们可以访问大小为 32 位的数据，该数据为寄存器 *rbx* 的低 32 位。因此，直接使用 *rbx* 便可访问该寄存器的全部 64 位数据。而使用 *bx* 与 *bl*，便可相应访问该寄存器的低 16 位与低 8 位数据。

另外，还需注意的一点是：我们可以通过不同的寄存器别名来读写同一寄存器不同位置上的数据。当某个指令需要重写寄存器的低 16 位或低 8 位数据时，寄存器中其他位上的数据不会被修改。而当指令需要重写寄存器低 32 位的数据时，高 32 位的数据会被同时复位，即置零。

听到这里，你可能觉得理解起来有些困难，不用担心，在课程的后面我还会多次介绍有关寄存器的内容。你可以先试着在 x86-64 平台上使用 GNU GCC 编译和运行下面这段代码。这里注意，在编译时不要为编译器指定任何优化参数。

复制代码

```
#include <stdio.h>
int main(void) {
  register long num asm("rax") = 0x100000000;
  asm("movl $0x1, %eax");
  // asm("movw $0x1, %ax");
  printf("%ld\n", num);
  return 0;
}
```

这样，你就可以看到，当指令作用于寄存器的不同部分时，CPU 对寄存器其他部分的影响。这里我们将值 0x100000000 放入寄存器 *rax* 中，在该 64 位值（long）对应的二进制编码中，其第 32 位被置位。第一句汇编指令将值 0x1 通过 `movl` 移动到 *rax* 寄存器的低 32 位；而第二句汇编指令将值 0x1 通过 `movw` 移动到 *rax* 寄存器的低 16 位。那么，通过这两种方式分别处理后的变量 num 的值是否相同呢？你可以自己进行实践，并在评论区留下答案。

随着课程的学习，你会看到这门课给出的示例代码中，存在着大量对这些通用寄存器的使用过程。而如何高效、有序地分配和使用寄存器，是编译器的重要任务之一，你会在后面的课程中了解到编译器在使用不同优化等级时对待寄存器的区别。

总结

讲到这里，今天的内容也就基本结束了。最后我们来一起总结下吧。

今天我主要介绍了常见数据量单位、汇编语言，以及指令集中寄存器的相关知识，希望这些基础知识能够为你接下来的学习提供一些帮助。

常见数据量单位包括位、字节和字。其中，一个字节等于 8 位，而字的大小则与具体的 CPU 体系结构紧密相关，常见大小可以是 32 位与 64 位。

汇编语言是一种低级编程语言，它用助记符的形式来描述程序对应机器指令的基本逻辑。由于它直接对应于 CPU 指令集之上，因此不具有可移植性。

寄存器是位于 CPU 芯片上的高速数据存储单元，根据功能，它可以被分为通用目的寄存器、状态寄存器等多种类型。x86-64 一共提供了 16 个通用目的寄存器，通过在汇编代码中使用不同的寄存器别名，我们可以快速访问这些寄存器中存放的数据。

思考题

这里，我们来一起做个思考题，巩固下今天的学习内容吧。

阅读下面的汇编代码，并尝试分析：当所有指令执行完毕时，寄存器 *eax* 中的值是多少？

复制代码

```
1 mov eax, 0x1
2 inc eax
3 sub eax, 10
4 xor eax, eax
5 add eax, 1
6 mul eax
```

这节选学课到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我交流讨论。如果这节课对你有帮助，也欢迎你把它分享给你的朋友或者同事。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 6 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 开篇词｜时至今日，如何更好地拥抱现代 C 语言？

下一篇 01｜快速回顾：一个 C 程序的完整生命周期

精选留言 (12)

写留言

ZR2021 神 置顶

2021-12-07

于老师，为啥main函数里的asm("movl $0x1, %eax");是将1给eax的，src 跟dst反过来了好像

作者回复: 这又是一个好问题！其实这里我们在代码中使用的是 AT&T 的写法，是一种默认被编译器广泛支持的内联汇编写法。当然，我们也可以做适当的调整来使用 Intel 写法，比如这样：

```
#include <stdio.h>
int main(void) {
  register long num asm("rax") = 0x100000000;
  asm(
    ".intel_syntax noprefix \n\t"
    "mov eax, 1\n\t"
    // "mov ax, 1\n\t"
    ".att_syntax"
  );
  printf("%ld\n", num);
  return 0;
}
```

但这种方式的问题在于，对于某些汇编器可能没有很好的兼容性。

共 2 条评论 > 6

权 麻

2021-12-06

验证重写64位寄存器低32位/低16位/低8位的代码中如果不是按照文中那样给寄存器赋值，而是任意赋值，建议大家将printf("%ld\n", num);改为printf("%lx\n", num);，输出16进制数，这样才能观察到正确的结果，否则输出10进制数，一开始没注意的话会发现结果很奇怪

...

展开

作者回复: 答案是正确的哈！

4

糊糊

2021-12-07

请教老师， 文中 mov ebx, 1 ，它所对应的机器指令代码为二进制值 bb 01 00 00 00 ，问题有两个：

1、汇编助记符 mov 是如何被翻译为二进制的？是通过在哪里查的表吗 ，如果是查的表，那表又存在哪里呢

2、 那根据机器码二进制能否推导出汇编代码呢？bb 01 00 00 00 ==》mov ebx, 1

展开

作者回复: 很好的问题！

第一个问题：每一个汇编指令都有其对应的组成结构，汇编器会根据助记符的名称进行相应的转换。具体的转换细节可以参考官方手册。比如对于 x86-64：https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

第二个问题：二进制状态下的机器指令识别可以通过每个指令对应的 OpCode 字节进行切分，然后再根据相应的结构解析出整体指令的结构。

共 3 条评论

2

dog_brother

2021-12-07

按老师的程序去执行，第一种（asm("movl $0x1, %eax");）结果是1，第二种（asm("movw $0x1, %ax");）结果是4294967297。

==========

```c
#include <stdio.h>...
```

展开

作者回复: 关于这个问题我可以先给点提示，然后你再思考下看看。第一个提示就是把第一行 printf 中的 “%d” 改成 “%ld” 就可以输出正确的结果；第二个提示是 printf 的调用会影响寄存器 rax 的值。

1

jack123
2021-12-07

xor eax，eax 相当于把eax置零了，最后+1，然后mul相乘 默认与eax相乘，最后还是为1

作者回复: 没错！

2

cc
2021-12-08

在 x86-64 架构下，CPU 指令集架构（ISA）中一共定义了 16 个通用目的寄存器，这些寄存器最大可以存放 4 个字，即 64 位长的数据。在汇编代码中
---

x86-64 下一个字长不就 64bit 了吗，为什么这里说 「4 个字，即 64 位长的数据」

展开

作者回复: 这是一个很好的发现！我们修改了文章，稍后会更新。

简单来讲就是：字这个概念会在多个地方的多种不同场景下使用。而文中我们介绍的字实际上是指可以体现 CPU 硬件特征的那个字，我一般会称它为硬件字。而在 CPU 指令集中，字也被用来作为衡量数据大小的单位。x86 架构由于历史原因，会将指令集中出现的 WORD 定义为固定 16 位的大小。所以实际文中提到的 4 个字你可以理解为指令字，它的单位是 WORD，4 个字即对应 64 位大小。

1

=
2021-12-07

老师好，请问Intel的寄存器在存储数据和读取数据时，也遵循小头优先吗？

作者回复: 其他 Intel CPU 不太清楚诶，在 x86-64 上是小端序的。

共 2 条评论 >

=
2021-12-07

eax异或后为0，然后再加1，因此eax中的内容变为1，最后与1相乘，结果仍然为1

作者回复: 正解！

Geek_828b39
2021-12-07

当某个指令需要重写寄存器的低 16 位或低 8 位数据时，寄存器中其他位上的数据不会被修改。而当指令需要重写寄存器低 32 位的数据时，高 32 位的数据会被同时复位，即置零。
按照文章的描述，当执行asm("movl $0x1, %eax");后，num应该输出是1。当执行asm("movw $0x1, %ax");后应该输出0x100000001。但是实际都是输出1。老师，这是文章描述...
展开

作者回复: 可以简单描述下你的执行环境和代码吗？

共 5 条评论

一步
2021-12-07

怎么知道不同 型号 CPU 寄存器的名称的？ 比如我是 M1 Max 芯片的 CPU 使用 rax 报错提示 unknown register name 'rax' in asm
展开

作者回复: 每一类体系架构的 CPU 都有自己独立的 ISA，其中便描述了机器指令可以使用哪些寄存器。关于 M1 Max，你可以参考这里哈：https://developer.arm.com/documentation/ddi0487/ga

共 4 条评论

jack123
2021-12-07

第一个例子rax寄存器那里的置位0x100000000 是不是多了一个0 置到36位了

作者回复: 看了一下，应该没错的，是 1 后面 8 个 0 哈。

共 2 条评论

ROCKS

2021-12-07

在给定的示例中我多加了一步，我在给rax寄存器赋值后将值打印出来，想要与mov后的结果做比较。得到的结果与给定的示例不一样, mov后打印出来的结果都是1。是因为print函数执行的过程会清空寄存器吗？

#include <stdio.h>

int main()...

展开

作者回复: 这个是由于第二行 printf 函数的调用导致的，更多内容我们会在第 12 讲中详细介绍。概括下来是由于 x86-64 ABI 中对可变长参数调用时的规定。在这里，寄存器 rax 会用于保存使用到的向量寄存器的个数。

共 3 条评论